

Precise and Progressing Compositional Symbolic Execution

Dries Vanoverberghe
Frank Piessens

Report CW 582, April 2010



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Precise and Progressing Compositional Symbolic Execution

Dries Vanoverberghe
Frank Piessens

Report CW 582, April 2010

Department of Computer Science, K.U.Leuven

Abstract

Given a program and an assertion in that program, determining if the assertion can fail is one of the key applications of program analysis. Symbolic execution is a well-known technique for finding such assertion violations. It enjoys the following two interesting properties. First, symbolic execution is *precise*: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Second, it is *progressing*: if there is an execution that makes the assertion fail, it will eventually be found. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure.

Recently, *compositional* symbolic execution has been proposed. It improves scalability by analyzing each execution path of each method only once. However, proving precision and progress is more challenging for these compositional algorithms. This paper investigates under what conditions a compositional algorithm is precise and progressing (and hence a semi-decision procedure), and reports on the implementation of one such algorithm.

Keywords : compositional, symbolic execution, precision, progress.

Precise and Progressing Compositional Symbolic Execution: Extended Version

Dries Vanoverberghe* and Frank Piessens

Dries.Vanoverberghe, Frank.Piessens@cs.kuleuven.be

Abstract. Given a program and an assertion in that program, determining if the assertion can fail is one of the key applications of program analysis. Symbolic execution is a well-known technique for finding such assertion violations. It enjoys the following two interesting properties. First, symbolic execution is *precise*: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Second, it is *progressing*: if there is an execution that makes the assertion fail, it will eventually be found. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure.

Recently, *compositional* symbolic execution has been proposed. It improves scalability by analyzing each execution path of each method only once. However, proving precision and progress is more challenging for these compositional algorithms. This paper investigates under what conditions a compositional algorithm is precise and progressing (and hence a semi-decision procedure), and reports on the implementation of one such algorithm.

Key words: compositional, symbolic execution, precision, progress

1 Introduction

Given a program and an assertion in that program, determining whether the assertion can fail is one of the key applications of program analysis. There are two complementary approaches.

One can try to determine whether the assertion is *valid*, i.e. is satisfied in all executions of the program. This can be done using techniques such as type systems, abstract interpretation, or program verification. Such techniques are typically expected to be *sound*: if they report an assertion as valid, there will indeed be no execution that violates the assertion. However, these techniques suffer from false positives: they may fail to establish the validity of an assertion even if there is no execution that violates the assertion.

Alternatively one can look for counterexamples by trying to determine inputs to the program that will make the assertion fail. One important technique for this approach is symbolic execution [1], a well-known analysis technique to explore the execution traces of a program. The program is executed symbolically using logical symbols for program inputs, and at each conditional the reachability of both branches is checked

* Dries Vanoverberghe is a research assistant of the Fund for Scientific Research - Flanders (FWO)

using an SMT solver. When reaching the assertion, the analysis determines if it can find values for the symbolic inputs that falsify the assertion. Such a technique can not prove the validity of an assertion, but it has the advantage of avoiding false positives (a property that we will call *precision*). Obviously, sound and precise approaches are complementary. This paper focuses on precise algorithms, and more specifically on precise symbolic execution.

Thanks to many improvements to SMT solvers, symbolic execution has become an important technique, both in research prototypes [2–10] as well as in industrial strength tools [3, 4]. Recently, *compositional* symbolic execution [11, 12] attempts to further improve the scalability of symbolic execution. With compositional symbolic execution, each execution path of a method is only analyzed once. The results of this analysis are stored in a so-called *summary* of the method, and are reused by all callers of the method.

Traditional whole-program (non-compositional) symbolic execution has two interesting properties that are not necessarily maintained in the compositional case. First, as discussed above, symbolic execution is *precise*: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Proving precision for whole-program symbolic execution is relatively easy: one has to prove that symbolic execution correctly abstracts concrete executions, and that the SMT solver is sound and complete (which it can be for the class of constraints it needs to solve). Second, symbolic execution *makes progress* or is *progressing*: if there is an execution that makes the assertion fail, it will eventually be found. Therefore, there are no classes of programs where the analysis fails fundamentally. Again, making a symbolic execution algorithm progressing is relatively straightforward, for instance by making the algorithm explore the tree of possible paths through the program in a breadth-first manner. Since this tree is finitely-branching, a breadth-first exploration ensures that any node of the tree will eventually be visited. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure for the existence of counterexamples.¹

Although compositional symbolic execution is inspired by standard symbolic execution, the proofs of these important properties become much more challenging. In fact, some of the algorithms proposed recently are not necessarily semi-decision procedures. This paper develops proof techniques for showing precision and progress of compositional symbolic execution algorithms.

More specifically, this paper makes the following contributions:

- We create a formal framework for compositional symbolic execution, based on a small but powerful calculus.
- We show that any compositional symbolic execution algorithm based on this framework is *precise*.
- We give sufficient conditions for an algorithm to be *progressing*, and therefore be a semi-decision procedure.
- We report on an implementation of an algorithm that is precise and progressing, and hence is a semi-decision procedure.

¹ Note that precision is a soundness property, and progress is a completeness property, but we avoid the terms soundness and completeness on purpose to avoid confusion with soundness and completeness of verification algorithms or theorem provers.

For the purpose of investigating precision and progress, the assertion in the program is not relevant. What matters is whether the symbolic execution algorithm correctly enumerates all the reachable program states. Hence, for the rest of this paper, we will consider symbolic execution algorithms to be algorithms that enumerate reachable program states. Such an algorithm is precise if any program state that it enumerates is also reachable by the program. It is progressing if any program state reachable by the program is eventually enumerated.

The rest of this paper is structured as follows. First, in Section 2 we show by means of examples that precision and progress are hard to achieve for compositional algorithms. Then we introduce a small but powerful programming language in Section 3. Section 4 contains the compositional symbolic execution algorithm and creates a formal model of it based on transition systems. Next, we show that this algorithm is precise and progressing (Section 5) and report upon an implementation of this algorithm (Section 6). Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Motivation

Traditional symbolic execution explores paths through the program. Loops are unrolled, and method calls are inlined. If the program calls a given method several times, the execution paths in that method will be re-analyzed for each call. The key idea of compositional algorithms is to avoid this repeated analysis. Instead, execution paths are explored for each method independently. The results of this exploration are stored in a *method summary*. Method calls are no longer inlined: a method call is analyzed in one single step and the result is computed based on the summary of the target method. Compositional symbolic execution has been shown [11–13] to improve performance, but maintaining precision and progress is challenging.

2.1 Precision

Compositional symbolic execution creates two potential causes of imprecision. First, when there is insufficient information about the calling context of a method, then one might conclude that unreachable program locations are reachable. For example, the highlighted statement in the method *P2* in Figure 1 is unreachable in the current program because the method *P1* only calls *P2* with argument $x \neq 0$. However, if one would analyze *P2* independently of *P1*, the analysis might conclude that the highlighted statement is reachable. In other words, since reachability is a whole-program property, we need to maintain some whole-program state even in a compositional analysis. The example algorithm we discuss later will do so by maintaining an invocation graph.

Second, when a method returns and the analysis loses information about the relation between the arguments of the method and the return value, then the analysis might incorrectly conclude that a program location is reachable. For example, the highlighted statement in the method *P1* in Figure 1 is unreachable. When the analysis overapproximates the result of *P2* by the relation $result == 0 \vee result == 1 \vee result == -1$, then the highlighted location is reachable. To maintain precision, method summaries should not introduce such approximations.

```

int P1(int x) {
  if(x != 0){
    int r2 = P2(x);
    if(x > 0 && r2 != 1) return -1;
  }
  return 0;
}

int P2(int u) {
  if(u == 0) return 0;
  else if(u > 0) return 1;
  else return -1;
}

```

Fig. 1. Example program for precision

2.2 Progress

Non-compositional symbolic execution builds one global execution tree where leaf nodes represent either final program states, unreachable program states, or program states that require further analysis. Given a fair strategy to select such leaf nodes for further analysis, it is easy to show that the depth of the highest unexplored node keeps increasing and hence that any finite execution path will eventually be completely analyzed. This implies progress for non-compositional symbolic execution.

For compositional symbolic execution, the situation is more complex due to two reasons. First, as we discussed above, in order for method summaries to be precise, they must depend on the calling context. Hence, the discovery of a new call site for a method may increase the number of reachable points in the method and unreachable leaf nodes need to be reanalyzed taking into account the new calling context.

Secondly, when analyzing a method call, a compositional analysis relies on the summary of the target method for computing the return value. However, method summaries change over time when the analysis discovers new returns. As a consequence, nodes that were deemed unreachable based on the summary of the method must be reanalyzed when that method summary is updated.

The progress argument for non-compositional symbolic execution relies essentially on the fact that unreachable leaf nodes remain unreachable for the rest of the analysis. With compositional symbolic execution, this premise is no longer satisfied. Furthermore, it is impossible to guarantee that any finite execution path within the execution tree of a single method will eventually be completely analyzed. The program in Figure 2 provides an example of this phenomenon.

First, we explain the program: The two highlighted statements are both unreachable, and therefore the method *M1* returns 0 for any input. To understand this, two invariants are important: First, the method *M1* only calls the method *M2* with parameters $u = v$ with $u > 0$. Second, if the parameters u and v of *M2* are greater than zero, then *M2* returns the minimum of u and v .

Figures 3(a) and 3(b) show the execution trees of *M1* and *M2*. Each circle represents a case split in the program, and the corresponding condition is written on the upper-right corner. From a circle, the arc to the left (right) means that the condition is false (true). Squares are final nodes, and imply that the method returns with the return value written inside the square. Triangle denotes unreachable nodes.

Let $u_{i,j}$ be the analysis step that checks the j -th unreachable node of M_i , and f_i the sequence of analysis steps that explores the reachable part of the execution tree of

```

int  $M1$ (int  $x$ ) {
  while ( $x > 0$ ) {
    int  $y = M2(x, x)$ ;
    if ( $y < 0$ ) return  $-1$ ;
     $x--$ ;
  }
  return  $0$ ;
}

int  $M2$ (int  $u$ , int  $v$ ) {
  int  $w = 0$ ;
  while ( $u > 0$ ) {
    if ( $v \leq 0$ ) return  $-w$ ;
     $u--$ ;  $v--$ ;  $w++$ ;
  }
  return  $w$ ;
}

```

Fig. 2. Example program for progress

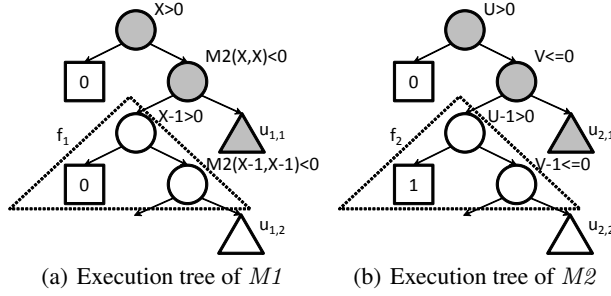


Fig. 3. Execution trees

M_i . The sequence f_1 causes a new invocation from $M1$ to $M2$ and therefore resets the unreachable nodes in $M2$. The sequence f_2 causes a new return and therefore resets the unreachable nodes in $M1$. Let $u_{i,2..}$ be the sequence $u_{i,2}, \dots, u_{i,n}$ where $u_{i,n}$ checks the the deepest unreachable node of M_i . Suppose we analyze according to the fair schedule $f_1, f_2, [u_{2,1}, f_1, u_{2,2..}, u_{1,1}, f_2, u_{1,2..}]^*$. Then the highlighted nodes in the execution trees, that are only at depth 3 in the tree will be of status needs-further-analysis an infinite number of times. Hence, the depth of the analysis never stays larger than 3, and a proof of progress fails.

This shows that progress for compositional symbolic execution cannot be proved by mimicking the proof for the non-compositional case on a per-method basis. In Section 5, we will propose an alternative technique to prove progress for compositional symbolic execution.

3 Programming language

In this section, we introduce a small intermediate language that is particularly well-suited for presenting compositional symbolic execution. It only retains the structure of the program that is essential: the structure of the control flow graph per procedure, and the calls and returns between procedures. The language focuses on sequential programs. Besides this restriction, all relevant more complicated language features can be translated to this core (e.g. parameters, return values or loops, ...). For simplicity of presentation, we also assume that the program does not contain (mutually) recursive

methods. This is not a restriction since it is possible to implement an interpreter that can handle a set of mutually recursive methods using an explicit stack data structure and loops. Our implementation (Section 6) supports recursion.

A program p is a tuple $\langle M_p, G_p, m_p^0 \rangle$ where M_p is a set of methods, G_p is a set of global variables and $m_p^0 \in M_p$ is a distinguished entry method. Each method definition m for the program p is a tuple $\langle L_m, N_m, \lambda_m, n_m^0 \rangle$ where L_m is a finite set of local variables disjoint from the global variables G_p , N_m is a finite set of program locations, $n_m^0 \in N_m$ is a distinguished entry node and $\lambda_m : N_m \rightarrow \text{Commands}_{m,p}$ maps each node to a command. The sets of local variables and nodes of different methods are disjoint.

A command c for the method m of the program p is either:

- An assignment **assign** x, e, n where $x \in L_m \cup G_p$, e is a side-effect free expression over $L_m \cup G_p$ and constants, and $n \in N_m$ is a program location. This command updates the value of the variable x , and continues in location n .
- A conditional **if** e, n_t, n_f where e is a side-effect free expression over $L_m \cup G_p$, and $n_t, n_f \in N_m$ are program locations. If the expression e evaluates to true (false) the execution continues in location n_t (n_f).
- A call **call** m_t, n where $m_t \in M_p$ is the target method and $n \in N_m$ is a program location. This command invokes the method m_t and continues in location n .
- A return **ret** returns from the current method.

For each variable v , $\mathcal{D}(v)$ represents the value domain of the variable. A valuation σ_V is a partial function that maps each variables $v \in V$ to a value $val \in \mathcal{D}(v)$. Each domain has a default element $\mathcal{D}_0(v)$, and the default valuation σ_V^d for a set of variables V maps each variable $v \in V$ to $\mathcal{D}_0(v)$.

An execution state $s \in S_p$ for the program p is tuple $\langle \sigma_G, \bar{f} \rangle$ where

- σ_G is the current valuation for G_p
- $\bar{f} \in F_p^*$ is a sequence of frames for p (sequences are either empty (nil) or a concatenation $h; \bar{t}'$ of a head h and a tail \bar{t}')

A frame $f \in F_p$ for the program p is a tuple $\langle m, n_m, \sigma_{L_m} \rangle$ where

- $m \in M_p$ is the current method.
- $n \in N_m$ is the current program location.
- σ_{L_m} is the current valuation for L_m

Figure 4 (in appendix) defines the operational semantics $\rightarrow \subseteq S \times S$, which gives an interpretation to the commands, and \rightarrow^* is its reflexive transitive closure. The premises **let** $x := y$ are not real conditions, they provide abbreviations for long expressions.

We use the following syntax for partial functions: Let $\text{Dom}(f)$ be the domain of the partial function f . The union of two partial functions f_1 and f_2 with disjoint domains is denoted by $f_1 \cup f_2$. Restriction of a partial function f to the domain D is denoted by $f|_D$. A singleton partial function $x \mapsto r$ maps the input x to a result r . The over-riding $f_1 \oplus f_2$ of a partial function f_1 by a partial function f_2 is the disjoint union of $f_1|_{\text{Dom}(f_1) \setminus \text{Dom}(f_2)}$ and f_2 . In addition, the evaluation $\text{eval}(\sigma_V, e)$ of an expression e

$$\begin{array}{c}
\frac{\lambda_m(n) = \text{assign } x, e, n' \quad \text{let } \sigma_T := (\sigma_G \cup \sigma_L) \oplus x \mapsto \text{eval}(\sigma_G \cup \sigma_L, e)}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \bar{f} \rangle \rightarrow \langle \sigma_T|_{G_p}, \langle m, n', \sigma_T|_{L_m} \rangle; \bar{f} \rangle} \text{ASSIGN} \\
\\
\frac{\lambda_m(n) = \text{if } e, n_t, n_f \quad \text{eval}(\sigma_G \cup \sigma_L, e) = \text{true}}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \bar{f} \rangle \rightarrow \langle \sigma_G, \langle m, n_t, \sigma_L \rangle; \bar{f} \rangle} \text{COND-T} \\
\\
\frac{\lambda_m(n) = \text{if } e, n_t, n_f \quad \text{eval}(\sigma_G \cup \sigma_L, e) = \text{false}}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \bar{f} \rangle \rightarrow \langle \sigma_G, \langle m, n_f, \sigma_L \rangle; \bar{f} \rangle} \text{COND-F} \\
\\
\frac{\lambda_m(n) = \text{call } m_t, n' \quad \lambda_m(n) = \text{ret} \quad \bar{f} \neq \text{nil}}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \bar{f} \rangle \rightarrow \langle \sigma_G, f_{m_t}^0; \langle m, n, \sigma_L \rangle; \bar{f} \rangle \text{ CALL } \langle \sigma_G, \langle m, n, \sigma_L \rangle; \bar{f} \rangle \rightarrow \langle \sigma_G, \bar{f} \rangle} \text{RET}
\end{array}$$

Fig. 4. Operational semantics

in a valuation σ_V is defined by substituting each variable $v \in V$ by $\sigma_V(v)$ in e in the usual way.

The execution of a program p starts in the initial state $s_p^0 = \langle \sigma_{G_p}^0, f_{m_p^0}^0 \rangle$ with $f_{m_p^0}^0$ the initial frame for m_p^0 , and $\sigma_{G_p}^0$ the input valuation for the global variables G_p . The initial frame for a method m is $f_m^0 = \langle m, n_m^0, \sigma_{L_m}^d \rangle$.

A state s is reachable from a state s' if and only if $s \rightarrow^* s'$. A state s is reachable in a program pr (denoted as $\models \text{reach}(pr, s)$) if and only if s is reachable from the initial state s_{pr}^0 of the program.

4 Compositional symbolic execution

Symbolic execution [1] is a technique to explore the execution paths of a program under all possible inputs. Instead of using a concrete input, the execution of the program is started with symbols representing arbitrary values. As a result, the values in the symbolic execution state are symbolic expressions that depend on the input symbols. Symbolic interpretation lifts the interpretations of commands to symbolic states.

For each execution path, symbolic execution constructs a *path condition*, a constraint in function of the input symbols that characterizes when an input follows that path. At a branch with condition C , the result of concrete execution is statically unknown: Either C is true and execution continues with the true-branch, or C is false and the false-branch is taken. Therefore, symbolic execution splits the set of inputs in two new sets: one where C is added to the path condition and one where the negation of C is added to the path condition. To check reachability, i.e. whether there is an execution that follows a path, a constraint solver checks satisfiability of the path condition. Traditional non-compositional symbolic execution thus computes a global partition of the input space.

The difference between compositional and traditional symbolic execution is in the treatment of method calls and returns. In traditional symbolic execution, a call adds a new symbolic frame for the target method and continues execution until a return command pops this frame. Therefore, when a method is called twice, a path through the method is computed twice, even if it is guaranteed to follow the same path. With compositional symbolic execution, each method is analyzed in isolation. This results in a partition for each method (also called the summary). Instead of performing a real call, compositional symbolic execution uses the summary of the target method to compute the effect on the symbolic state.

To show that compositional symbolic execution is a semi-decision procedure, it is convenient to model the algorithm as a transition system $a \Rightarrow a'$ (and \Rightarrow^* its reflexive transitive closure) which starts in an initial analysis state a^0 . Non-terminating runs of the algorithm can be truncated after any number of transitions. In addition, the predicate $\vdash^a \text{reach}(p, s)$ denotes that the analysis concludes the reachability of the state s in the program p in an analysis state a .

Such a transition system is precise if and only if the conclusion in any reachable analysis state is sound²:

Definition 1 (Precision). *For each program pr , concrete state s , and analysis state a such that $a_{pr}^0 \Rightarrow^* a$, $\vdash^a \text{reach}(pr, s)$ implies $\models \text{reach}(pr, s)$.*

Obviously, compositional symbolic execution is not complete³ in any reachable analysis state and due to undecidability this is even impossible. However, the analysis incrementally discovers more and more reachable states. This incremental nature is captured by monotonicity:

Definition 2 (Monotonicity). *For each program pr , concrete state s , and analysis states a, a' such that $a \Rightarrow a'$, if $\vdash^a \text{reach}(pr, s)$ then $\vdash^{a'} \text{reach}(pr, s)$.*

For a monotonous analysis, progress is the next best thing with respect to completeness: for any reachable concrete state, eventually there is an analysis state that is complete for that concrete state:

Definition 3 (Progress). *For each program pr and each concrete state s , if $\models \text{reach}(pr, s)$ then for all analysis states a' such that $a_{pr}^0 \Rightarrow^* a'$ there exists an analysis state a such that $a' \Rightarrow^* a$ and $\vdash^a \text{reach}(pr, s)$. In other words, for each reachable concrete state s , there always eventually is an analysis state that concludes s reachable.*

When an analysis is precise, monotonous and progressing, it is a semi-decision procedure.

4.1 Overview

The analysis state maintains a summary per method, which is a set of leaf nodes of the (partially explored) execution tree of the method. A leaf node $\langle \text{stat}, \nu, pc \rangle$ contains:

² sound as a bugfinder, i.e. any state which is concluded reachable is truly reachable

³ complete as a bugfinder

- A status $stat$, which is either unknown, finished or unreachable,
- A symbolic execution state ν ,
- A path condition pc .

The path condition defines the inputs (i.e. the values of the global variables) that will drive the execution of the method along this path. The symbolic execution state represents the state of execution after executing the path. The status indicates whether (a) the path is a complete path through the method, i.e. the method returns after this path (finished status) (b) the path is unreachable (unreachable status) (c) further exploration of continuations of this path are needed (unknown status). Symbolic execution states are defined like concrete execution states, except that all valuations are symbol valuations i.e. any variable has a symbolic expression instead of a concrete value.

The summaries only maintain per-method information. As we have shown in Section 2, it is necessary to maintain some whole program information in order to be precise. In particular, it is important to precisely track method call sites and returns that are reachable.

Initially, only the main method is reachable. As the analysis progresses, any call that is discovered is stored in an invocation graph. This graph is represented as a set of invocations, where each invocation is a tuple $\langle m_s, m_t, \varsigma_G, pc \rangle$. The methods m_s and m_t are the source and target methods, and ς_G and pc are the symbolic values of the global variables and the path condition at the moment of the invocation. Reachability checking will use the information in the call graph to decide whole-program reachability.

To support discovery of new returns efficiently, we model the possible return values of method calls using logic function symbols. The symbolic execution of a call will be defined in terms of these function symbols. The interpretations of the function symbols are constructed using the current summary. As the analysis progresses, they become precise for more and more inputs. We discuss this in more detail in Section 4.2.

In addition, the analysis tracks all reachable program states it has enumerated. For this purpose, the analysis state contains a set of leaf-nodes that succeeded the reachability check for each method. Based on this information, reachability conclusion is defined. If a leaf-node $\langle stat, \nu, pc \rangle$ is in the reachable set of the method m , then any concretization of its symbolic state ν with global variables satisfying pc is reachable in m . A state s is concluded reachable (denoted $\vdash^a reach(pr, s)$) if and only if either

- s is reachable in the entry method m_{pr}^0 .
- there is a state s' such that $\vdash^a reach(pr, s')$ and s' calls m and s is reachable in m .

Usually, one is only interested whether a point in a program is reachable (e.g. a location n in a method m). Therefore, implementations often store reachable program points instead of leaf-nodes or avoid the reachability set completely by reporting an error when reaching a distinguished error-location. However, reachability conclusion of arbitrary states is essential for inductive invariants that enable the precision and progress proofs.

Finally, an analysis state can be defined as a tuple $\langle sum, invs, rs \rangle$ where

- sum is a function that maps each method m to a set of leaf-nodes (its summary).
- $invs$ is a set of invocations.
- rs is a function that maps each method to a set reachable leaf-nodes.

$$\begin{aligned}
a &::= \langle sum, invs, rs \rangle \\
i &::= \langle m_s, m_t, \varsigma_G, pc \rangle \\
\epsilon &::= \langle stat, \nu, pc \rangle \\
\varrho &::= \langle m, n, \varsigma_{L_m} \rangle \\
\nu &::= \langle \varsigma_{G_p}, \varrho \rangle
\end{aligned}$$

Fig. 5. Definition of analysis states

Figure 5 summarizes all definitions with respect to analysis states.

Figure 6 defines the initial analysis state a_p^0 , where the invocation graph and the sets of reachable leaf-nodes are empty. For each method, the summary starts with one leaf-node with unknown status, path condition *true* and a symbolic execution state at the entry of the method, where the value of all global variables contains a new symbol.

$$\begin{aligned}
a_p^0 &::= \langle sum_p^0, \emptyset, \emptyset \rangle \\
sum_p^0 &::= \bigcup_{m \in M_p} m \mapsto \{\epsilon_m^0\} \\
rs_p^0 &::= \bigcup_{m \in M_p} m \mapsto \emptyset \\
\epsilon_m^0 &::= \langle unk, \nu_m^0, true \rangle \\
\nu_p^0 &::= \langle \varsigma_{G_p}^0, \varrho_{m_p^0}^0 \rangle \\
\varrho_m^0 &::= \langle m, n_m^0, \sigma_{L_m}^d \rangle
\end{aligned}$$

Fig. 6. Initial analysis state

The high level overview of one step of the compositional symbolic execution algorithm is shown in Figure 7. During each step, the algorithm chooses a method m and an leaf-node $\epsilon \in sum_a(m)$ with unknown status. Then, the algorithm checks whether there exists an input $\sigma_{G_p}^0$ such that the execution enters the method m and the global variables satisfy the path condition pc_ϵ of ϵ ($Check(a, m, \epsilon.pc)$). If there is no such input, the status of the ϵ is changed to unreachable. Otherwise, ϵ is added to the set of reachable leaf nodes of m , and symbolic execution continues with the interpretation ($SyInt(a, m, \epsilon)$) of the symbolic state ν_ϵ of ϵ . When symbolic interpretation finishes, it returns a set of new equivalence leaf-nodes, and the current leaf-node ϵ is replaced by the new leaf-nodes ($ReplaceLeaf$). All method calls in this algorithm are guaranteed to terminate, and therefore one step of the algorithm always terminates.

We now zoom in on some aspects of the algorithm that are of importance for precision and progress.

4.2 Symbolic interpretation

Figure 8 shows the symbolic interpretation rules for the language of Section 3. Each rule is structured as follows:

$$\frac{\lambda_m(n) = \dots}{a \parallel m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a' \parallel m, \pi}$$

```

AnalysisState Step(AnalysisState a) {
  (m, ε) = Choose(a);
  if(Check(a, m, ε.pc)) {
    a' = AddReachable(a, m, ε);
    (a'', π) = SyInt(a', m, ε);
    return ReplaceLeaf(a'', m, ε, π);
  } else {
    return MarkUnreach(a, m, ε);
  }
}

```

Fig. 7. High level algorithm of symbolic execution

It starts in an analysis state a with a chosen method m and leaf-node $\langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle$. As a result, it potentially changes the analysis state to a' and it returns a new set of leaf-nodes π .

$$\begin{array}{c}
\frac{\lambda_m(n) = \text{assign } x, e, n' \quad \text{let } \varsigma_T := (\varsigma_G \cup \varsigma_L) \oplus x \mapsto eval(\varsigma_G \cup \varsigma_L, e)}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a \| m, \{ \langle unk, \langle \varsigma_G, \langle m, n', \varsigma_T|_{L_m} \rangle \rangle, pc \rangle \}} \text{ ASSIGN} \\
\\
\frac{\lambda_m(n) = \text{if } e, n_t, n_f \quad \text{let } b := seval(\varsigma_G \cup \varsigma_L, e) \quad \text{let } \epsilon_1 := \langle unk, \langle \varsigma_G, \langle m, n_t, \varsigma_L \rangle \rangle, pc \wedge b \rangle \quad \text{let } \epsilon_2 := \langle unk, \langle \varsigma_G, \langle m, n_f, \varsigma_L \rangle \rangle, pc \wedge \neg b \rangle}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a \| m, \{ \epsilon_1, \epsilon_2 \}} \text{ COND} \\
\\
\frac{\lambda_m(n) = \text{call } m_t, n' \quad \text{let } \varsigma'_G := \bigcup_{v \in G_p} v \mapsto rv_{m,v}(\varsigma_G) \quad \text{let } \epsilon := \langle unk, \langle \varsigma'_G, \langle m, n', \varsigma_L \rangle \rangle, pc \wedge rc_m(\varsigma_G) \rangle \quad \text{let } invs'_a := invs_a \cup \{ \langle m, m_t, \varsigma_G, pc \rangle \} \quad \text{let } a' := rec(\langle sum_a, invs'_a, rs_a \rangle, m_t)}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a' \| m, \{ \epsilon \}} \text{ CALL} \\
\\
\frac{\lambda_m(n) = \text{ret} \quad \text{let } a' := rer(a, m)}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a' \| m, \{ \langle fin, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \}} \text{ RET}
\end{array}$$

Fig. 8. Interpretation rules

As pointed out in the previous section, the analysis uses uninterpreted function symbols to support discovery of new returns as the analysis progresses. The algorithm models the effect of the method m on the global variable v as an uninterpreted function symbol $rv_{m,v}$. When a method m is called with global variables ς_{G_p} , then the function application $rv_{m,v}(\varsigma_{G_p})$ models the value for the global variable v after executing m .

In addition, the method summaries are *partial*: there is no information about unexplored paths through a method. To deal with this, the algorithm models the set of global variable valuations that follow a finished path as an uninterpreted predicate rc_m .

During reachability checking, the algorithm computes the interpretation for the uninterpreted symbols using the method summaries (Figure 9) and replaces them using substitution (e.g. $int(a, pc)$ substitutes all uninterpreted symbols by substituting them for their interpretation).

$$\begin{aligned}
interps(sum) &= \bigcup_{m \in M_p} interps(m, \{\epsilon \mid \epsilon \in sum(m), stat_\epsilon = fin\}) \\
interp(m, \pi) &= rc_m \mapsto interp_{rc}(m, \pi) \cup (\bigcup_{v \in G_p} rv_{m,v} \mapsto interp_{rv}(m, v, \pi)) \\
interp_{rc}(m, \pi) &= \bigvee_{\langle fin, \nu, pc \rangle \in \pi} pc \\
interp_{rv}(m, v, \emptyset) &= \mathcal{D}_0(v) \\
interp_{rv}(m, v, \langle fin, \nu, pc \rangle \cup \pi) &= ite\ pc\ \varsigma_{G_\nu(v)}\ interp_{rv}(m, v, \pi)
\end{aligned}$$

Fig. 9. Interpretation of uninterpreted function symbols

For precision, it is essential that the interpretation of the uninterpreted symbols is precise: Whenever the return condition $rc_m(\sigma_{G_p})$ is true, the execution of the method m starting with global variables σ_{G_p} eventually reaches a return command, and each global variable v must equal $rv_{m,v}(\sigma_{G_p})$.

Definition 4 (Precision of interpretations). For each program p , each analysis state a , the interpretations $interps(sum_a)$ are precise if and only if for each method $m \in M_p$ and each global input valuation σ_{G_p} that satisfies $int(a, rc_m(\sigma_{G_p}))$, $\langle \sigma_{G_p}, f_m^0 \rangle \rightarrow^* \langle \sigma'_{G_p}, \langle m, n, \sigma_L \rangle; nil \rangle$ where $\lambda_m(n) = \mathbf{ret}$ and $\sigma'_{G_p}(v) = int(a, rc_{m,v}(\sigma_{G_p}))$ for any variable $v \in G_p$.

Definition 5 (All reachable). In an analysis state a , all concrete states on the execution run $s \rightarrow^* s'$ are reachable (denoted $allReach(a, s \rightarrow^* s')$) if and only if $\vdash^a reach(pr, s)$ and $s = s'$ or $s \rightarrow s'$ and $allReach(a, s' \rightarrow^* s')$.

Definition 6 (Restricted completeness of interpretations). For each program p , each analysis state a , the interpretations $interps(sum_a)$ are restricted complete if and only if for all states $s = \langle \sigma_{G_p}, f_m^0 \rangle$ and $s' = \langle \sigma'_{G_p}, \langle m, n, \sigma_L \rangle; nil \rangle$ such that $s \rightarrow^* s'$, $\lambda_m(n) = \mathbf{ret}$ and $allReach(a, s \rightarrow^* s')$, the return condition $int(a, rc_m(\sigma_{G_p}))$ is satisfied and $\sigma'_{G_p}(v) = int(a, rc_{m,v}(\sigma_{G_p}))$ for any variable $v \in G_p$.

The treatment of assignment and branches is similar to the treatment in non-compositional symbolic execution: For an assignment, symbolic interpretation performs the same operation but on symbolic expressions instead of concrete values. For branches, symbolic interpretation creates a new leaf-node for each branch and conjoins the branch condition or its negation to the path condition.

The rule call creates a new leaf node where the return condition is added to the path condition, and the return values are used to update the global variables. As mentioned in Section 2, some leaf-nodes can become reachable by performing a call, and

progress requires that all such leaf-nodes are reconsidered. The algorithm conservatively reconsiders all unreachable leaf-nodes of methods that are transitively reachable in the invocation graph by marking them as unknown (using the function $rec(a, m)$, defined more precisely in Appendix).

The return rule marks the unknown leaf-node as finished, and thereby the interpretations of the current method change. In addition, the return rule marks all unreachable leaf-nodes that depend on the return condition as unknown again (using the function $rer(a, m)$, also defined in Appendix). This is again essential to maintain progress.

For precision, the symbolic interpretation algorithm must maintain precision of the leaf-nodes, i.e. if an input is a member of a leaf-node, then the execution starting with that input eventually reaches the concretization of the symbolic state (the symbolic state after replacing the input symbols with the concrete input). In addition, all invocations $\langle m_s, m_t, \varsigma_G, pc \rangle$ in the invocation graph must be precise: If an input satisfies the condition pc , then the execution of m_s starting with that input reaches a call to the method m_t and the global variables are the concretization of ς_G .

In the following definition, we assume that $conc(\nu_\epsilon, \sigma_{G_p})$ substitutes the input symbols $\varsigma_{G_p}^0$ with σ_{G_p} :

Definition 7 (Precision of leaf-nodes). *Under an analysis state a , a leaf-node $\epsilon \in sum_a(m)$ is precise if and only if any global input valuations σ_{G_p} satisfying $int(a, pc_\epsilon)$, $\langle \sigma_{G_p}, f_m^0 \rangle \rightarrow^* conc(\nu_\epsilon, \sigma_{G_p})$ i.e. when starting the execution with the input σ_{G_p} and the initial frame for the method m , the execution reaches the concretization of the symbolic state ν_ϵ . The summaries sum_a are precise if any leaf-node ϵ of the summary $sum_a(m)$ of any method m is precise.*

For progress, it is essential that symbolic interpretation maintains *totality* of the summaries. A reachable concrete state s is on the frontier if all predecessors in the execution to s are concluded reachable, but s is not concluded reachable. The summaries are total if any concrete state s on the frontier is a concretization of some unknown leaf-node ϵ in the summary of some method m . Informally, this is a kind of completeness guarantee for symbolic interpretation. For any concrete state on the frontier, the analysis can make the “right” choice. Totality implies that leaf-nodes may not be marked unreachable if *Check* succeeds in the current analysis state. For this reason, the call and return rules need to reconsider some unreachable leaf-nodes.

Definition 8 (Frontier). *In an analysis state a , a concrete state s is on the frontier (denoted $front(pr, a, s)$) if and only if there exists a reachable concrete state s' such that all states on the execution $s_{pr}^0 \rightarrow^* s'$ are reachable, and $s' \rightarrow s$ and $\vdash^a reach(pr, s)$ is false.*

Definition 9 (Totality of summaries). *In an analysis state a , the summaries sum_a are total if and only if for any concrete state s on the frontier, there exists an unknown leaf-node $\epsilon \in sum_a(m)$ in the summary of some method m such that there is a global valuation σ_{G_p} satisfying pc_ϵ and s is a concretization of ν_ϵ .*

In addition, symbolic interpretation also maintains precision and totality of the invocations.

Definition 10 (Precision of invocations). Under an analysis state a , an invocation $\langle m_s, m_t, \varsigma_G, pc \rangle \in \text{invs}_a$ is precise if and only if for all global input valuations σ_{G_p} , if σ_{G_p} satisfies $\text{int}(a, pc)$ then $\langle \sigma_{G_p}, f_{m_s}^0 \rangle \rightarrow^* \langle \sigma_{G'_p}, \langle m, n, \sigma_L \rangle \rangle$ where $\lambda_m(n) = \text{call } m_t, n'$ and $\sigma_{G'_p}$ is the concretization of ς_G with input σ_{G_p} . In other words, when starting the execution with the input σ_{G_p} and the initial frame for the method m_s , the execution reaches an invocation of the method m_t where the global variables are the concretization of ς_G with input σ_{G_p} .

Definition 11 (Totality of invocations). In an analysis state a , the invocations invs_a are total if and only if for any method m and any concrete state s such that $\text{allReach}(a, \langle \sigma_{G_p}, f_m^0 \rangle \rightarrow^* s)$, if $s = \langle \sigma_{G'_p}, \langle m, n, \sigma_L \rangle \rangle$ and $\lambda_m(n) = \text{call } m_t, n'$, then there exists an invocation $\langle m, m_t, \varsigma_G, pc \rangle \in \text{invs}_a$ such that σ_{G_p} satisfies $\text{int}(a, pc)$ and $\sigma_{G'_p}$ is the concretization of ς_G with σ_{G_p} .

4.3 Reachability checking

Finally, to check reachability ($\text{Check}(a, m, pc)$), the algorithm globalizes the path condition pc based on the invocation graph invs_a , substitutes the symbols $\varsigma_{G_p}^0$ by their interpretation $\text{interps}(\text{sum}_a)$, and uses an SMT-solver to check the satisfiability of the resulting constraints. The globalization $\text{glob}_p(a, m, pc)$ globalizes the constraint pc in the context of m using the invocation graph invs_a and is defined inductively as follows:

- If $m = m_p^0$ then $\text{glob}_p(a, m, pc) = pc$
- If $m \neq m_p^0$ then $\text{glob}_p(a, m, pc) = \bigvee_{\langle m_s, m, \varsigma_G, pc', d \rangle \in \text{invs}_a} \text{glob}_p(a, m_s, pc' \wedge pc[\bigcup_{v \in G_p} \varsigma_{G_p}^0(v) \mapsto \varsigma_{G_p}(v)])$

In the absence of recursion, the invocation graph is cycle free, and the inductive definition is well-founded.

For precision, it is important that $\text{Check}(a, m, pc)$ only returns true when there is a reachable state s where the execution enters m and the global variables satisfy pc (precision of Check). This follows from precision of the leaf-nodes, the precision of the interpretations and the soundness of the SMT-solver as a satisfiability checker.

Definition 12 (Precision of reachability check). Reachability checking (Check) is precise if and only if for each program p , each reachable analysis state a , each method $m \in M_p$ and each condition pc , if $\text{Check}(a, m, pc) = \text{true}$ then there exists global input valuation σ_{G_p} such that m is invoked with global variable valuation σ_{G_p} and $\text{int}(a, pc)[\theta_p^0(\sigma'_{G_p})]$ holds.

The contrary is not the case: If there is an execution that enters m where the global variables satisfy pc , $\text{Check}(a, m, pc)$ need not return true because this execution might follow an unexplored path through some method. For progress, it is only necessary that $\text{Check}(a, m, pc)$ holds if the execution that enters m where the global variables satisfy pc only uses concrete states that are concluded reachable (*Restricted completeness*). This requires completeness of the SMT-solver as a satisfiability checker.

Definition 13 (Restricted-Completeness of reachability check). *Reachability checking (Check) is restricted-complete if and only if for each program p , each reachable analysis state a , if there exists a concrete state s such that $\text{allReach}(a, s_p^0 \rightarrow^* s)$ and s calls m with the global variable valuation σ_{G_p} satisfying the condition pc , then $\text{Check}(a, m, pc) = \text{true}$.*

5 Properties

In this section, we show that the algorithm of Section 4 is precise, and we show that it is also progressing as long as the choices are fair.

In Section 4, we defined the key invariants that enable precision and progress. First, we show that the algorithm maintains these properties. We say an analysis state is valid if it satisfies its invariants:

Definition 14 (Validity of analysis state). *An analysis state a is precise if and only if*

1. *Each leaf-node ϵ of the summary $\text{sum}_a(m)$ of each method m is precise.*
2. *Each finished leaf-node ϵ of the summary $\text{sum}_a(m)$ of each method m is returning from m .*
3. *The invocation graph invs_a is cycle free.*
4. *All invocations $\text{inv} \in \text{invs}_a$ are precise.*
5. *Each leaf-node ϵ in rs_a is precise, and globally reachable: there exists a global variable valuation σ_{G_p} such that m is invoked with valuation σ_{G_p} that satisfies $\text{int}(a, pc)$. (where m is active method of the symbolic state of ϵ).*
6. *The summaries sum_a are total.*
7. *The invocation graph invs_a is total.*
8. *For each all-reachable concrete state s that returns from m , $\text{sum}_a(m)$ contains a finished node ϵ such that s is a concretization of v_ϵ .*

Lemma 1 (Precision of interpretations). *For any program p and valid analysis state a , the interpretations are precise.*

Proof. For any valid analysis state, the invocation graph is cycle free. Therefore, we can do induction on the depth of the invocation graph.

- If the method m does not invoke another method, the proof is straightforward from the definition of the interpretations and the precision of the summary of m .
- Otherwise, the methods that can be called have a smaller depth in the invocation graph. The induction hypothesis implies that the interpretations of those methods are correct. The interpretation $\text{int}(a, pc)$ of a constraint pc substitutes the uninterpreted function symbols from these methods with their interpretation. Using the definition of the interpretations and the validity of the summary of m , we can again prove that the interpretation is precise.

Lemma 2 (Precision of reachability checking). *For any program p and valid analysis state a , the reachability checking is precise.*

Proof. By induction on the depth of the invocation graph.

- Base case: depth is zero, and therefore $m = m_p^0$. Since the interpretations are precise, everything depends on the ability of the SMT-solver to find a satisfying assignment. Therefore, if the smt-solver is sound as a constraint solver, i.e. the solver only returns true if there actually exists a solution, then the lemma holds.
- Using the induction hypothesis and the precision of the invocations.

Lemma 3 (Restricted completeness of interpretations). *For any program p and valid analysis state a , the interpretations are restricted complete.*

Proof. By induction on the depth of the invocation graph.

- Depth is zero, and therefore $m = m_p^0$. By validity, there is a finished leaf-node $\epsilon \in \text{sum}_a(m)$ for each all-reachable concrete state s that returns from m such that s is a concretization of ν_ϵ . By the definition of *interp*, rc_m returns true, and $rv_{m,v}$ equals the concrete value for the global variables in s .
- Using the induction hypothesis and totality of the invocations.

Lemma 4 (Restricted completeness of reachability checking). *For any program p and valid analysis state a , the reachability checking is restricted complete.*

Proof. By induction on the depth of the invocation graph.

- Depth is zero, and therefore $m = m_p^0$. Using totality of the summaries, and restricted completeness of the interpretations. Therefore, if the smt-solver is complete (as a constraint solver), i.e. if there is a satisfying solution then the solver returns true, then the lemma holds.
- Using the induction hypothesis and restricted completeness of the interpretations.

Lemma 5 (Validity of the initial analysis state). *For any program p , the initial analysis state a_p^0 is valid.*

Proof. Follows immediately from the definition of the initial analysis state.

Lemma 6 (Maintenance of validity). *For any program p , and analysis states a, a' , if a is valid and $a \Rightarrow a'$, then a' is valid.*

Proof. First, the method CHOOSE chooses a method m and an unknown node $\epsilon = \langle \text{unk}, \nu, pc \rangle \in \text{sum}_a(m)$. If no such equivalence node can be found in any method, the analysis is completed. There is no a' such that $a \Rightarrow a'$, so the theorem trivially holds.

If a method and unknown node has been selected, analysis continues with reachability checking. Suppose $\text{Check}(a, m, pc_\epsilon) = \text{false}$, then there is no global input valuation σ_{G_p} that satisfies $\text{int}(a, pc)[\theta_p^0(\sigma_{G_p})]$ and reaches m . The algorithm replaces ϵ with an unreachable class. This unreachable class is valid since the unknown class was valid, and the path condition and execution depth remain the same. In addition, it is impossible to violate totality of the summaries since check is restricted complete.

If $\text{Check}(a, m, pc_\epsilon) = \text{true}$ then there exists a global input valuation σ_{G_p} satisfying $\text{int}(a, pc)$ such that m is globally reachable. The algorithm refines the leaf-node and executes one or more steps of symbolic interpretation. In what follows, we show that one step of symbolic interpretation results in a new, valid analysis state. The same

result for multiple steps can be obtained by induction on the amount of steps. The added reachability element is also precise (by the precision of ν_ϵ) and globally reachable (by the precision of the reachability check).

The remainder of the proof is a case split on the symbolic interpretation rule:

ASSIGN The assign rule only updates the symbolic state, and therefore the members of the leaf-nodes class remain the same. Since ν_ϵ is precise, all valuations lead to a concrete state s where the projected execution rule ASSIGN applies. The symbolic interpretation rule applies the same mutation to the symbolic state, and is therefore again precise. In addition, if sum_a was total then the new summaries are also total, since all members of ϵ are now member of the new leaf node.

COND The argument for conditional branches is similar as for assignment except that there are now two new nodes ϵ_1 and ϵ_2 . By precision of ϵ , all executions lead to a concrete state s where either COND-T or COND-F applies. When COND-T applies, the input valuation becomes a member of ϵ_1 and ϵ_1 is again precise. Alternatively, when COND-F applies, the input valuation becomes a member of ϵ_2 and ϵ_2 is again precise. In addition, if sum_a was total then the new summaries are also total, since all members of ϵ are now member of ϵ_1 or ϵ_2 .

CALL The call rule creates a new leaf-node ϵ' that represents the symbolic state after returning from the target method. Since ϵ was precise, and the interpretations are precise, ϵ' is also precise.

In addition, the new invocation is added to the invocation graph and the function *rec* re-evaluates unreachable nodes. Marking unreachable nodes as unknown does not affect the precision of the nodes. Since all unreachable nodes of all methods that are reachable in the invocation graph are re-evaluated, the new summaries are total again. All inputs that previously were members of ϵ are now either reachable, or a member of some unknown class.

Because the program is not (mutually) recursive, the new invocation does not introduce cycles. In addition, by precision of ϵ , the new invocation is also precise.

RET The ret rule marks the unknown node as finished. The members of the equivalence class remain the same, and the symbolic state remains precise.

In addition, RET re-evaluates all unreachable nodes that depend on the return of m . All unreachable nodes that can become reachable are re-evaluated. Therefore, the new summaries are total again.

Corollary 1 (All reachable analysis state are valid). *For any program pr , and any analysis state a such that $a_{pr}^0 \Rightarrow^* a$, a is valid.*

Precision follows immediately from validity, since each leaf node in the reachability set is precise and *Check* succeeds.

Theorem 1 (Precision). *The algorithm is precise.*

The argument for progress is more complicated. First, we show that compositional symbolic execution is monotonous.

Theorem 2 (Monotonicity). *For each program pr , concrete state s , and analysis states a, a' such that $a \Rightarrow a'$, if $\vdash^a reach(pr, s)$ then $\vdash^{a'} reach(pr, s)$.*

Proof. Follows from the fact that (a) \Rightarrow never removes reachable leaf-nodes ($rs_a \subseteq rs_{a'}$). (b) \Rightarrow never removes invocations ($invs_a \subseteq invs_{a'}$).

Since the search tree of the algorithm is potentially infinite, monotonicity is not sufficient to find all reachable states: The algorithm might get stuck exploring only a subspace of the program. Fortunately, this can not happen if the analysis is *fair*, i.e. if each unknown class is eventually chosen by the algorithm.

Definition 15 (Fairness). *An application strategy of the compositional symbolic execution algorithm is fair if and only if for any analysis state a such that $a_{pr}^0 \Rightarrow^* a$, for any unknown class $\epsilon \in sum_a(m)$, the algorithm always eventually chooses $\langle m, \epsilon \rangle$.*

Finally, we show that compositional symbolic execution algorithm is progressing if it is fair. The proof shows a slightly stronger property, namely that there always eventually is an analysis state where all concrete states on the execution trace that reaches s are concluded reachable. This is essential since it gives a stronger induction hypothesis: we assume that all but the last concrete state s is concluded reachable and we show that the analysis always eventually reaches an analysis state where s is also concluded reachable. This hypothesis is necessary since a state might only be reachable from one invocation that has not yet been discovered, whereas its predecessor is already reachable based on another invocation. Together with totality of the summaries and restricted completeness of reachability checking, this allows a compact and intuitive proof for progress.

Theorem 3 (Progress). *If the compositional symbolic execution algorithm is fair, then it is progressing.*

Proof. By induction on \rightarrow^* .

Base step If s is the initial state, then $\vdash^a reach(pr, s)$ holds after applying the only possible analysis step.

Induction step If $s_{pr}^0 \rightarrow^* s'$ and $s' \rightarrow^* s$ and there always eventually is a reachable analysis state a' such that all concrete states from s_{pr}^0 to s' are concluded reachable in a' , then we must show that there always eventually is a reachable analysis state a such that $\vdash^a reach(pr, s)$. If $\vdash^{a'} reach(pr, s)$ already holds, then the proof is trivial.

1. First, we show that there exists an unknown class $\epsilon \in sum_{a'}(m)$ such that $Check(a', m, pc_\epsilon) = true$ and s is a concretization of ϵ . This means that if we choose $\langle m, \epsilon \rangle$, then the state s will become reachable in the next analysis state. This follows from the fact that the summaries are total, and restricted completeness of check.
2. By fairness, there always eventually exists a reachable analysis state a'' such that $\langle m, \epsilon \rangle$ has not been chosen yet, and is chosen in the next analysis step. Since $\langle m, \epsilon \rangle$ has not been chosen, it must still be in the summary of m ($\epsilon \in sum_{a''}(m)$). Because invocations are never removed ($invs_a \subseteq invs_{a''}$), the method $Check$ is monotonous and $Check(a'', m, pc_\epsilon) = true$. Therefore, if $\langle m, \epsilon \rangle$ is chosen in $a'' \Rightarrow a$, then $\vdash^a reach(pr, s)$.

6 Implementation

To show that the algorithm in Section 4 provides similar speedup as other compositional symbolic execution tools [12, 11], we have implemented one instantiation of the framework where the programming language is the intermediate language of the .NET platform [14]. For parsing bytecode, we use the Mono.Cecil [15] library and as constraint solver we use Z3 [16].

Our tool is based on dynamic symbolic execution, a variant of symbolic execution where the program is executed with real inputs and monitored during execution to build a symbolic representation on the side. Dynamic symbolic execution prevents false positives because it detects and reports preciseness problems at runtime. Whenever the real execution does not follow the intended path the tool reports it. This was useful to debug the symbolic interpretation rules before we proved precision.

We applied both the compositional and the interprocedural version of our tool on the example *IncDec* described in [12] until all paths through the program were analyzed. We measured the execution time (Figure 10(a)) and the amount of queries (Figure 10(b)) in function of the bound N . All experiments were conducted on an Intel Core 2 Duo T7500 (2.2 GHz) with 4Gb of memory. We repeated each experiment 10 times and report the averages results of all experiments.

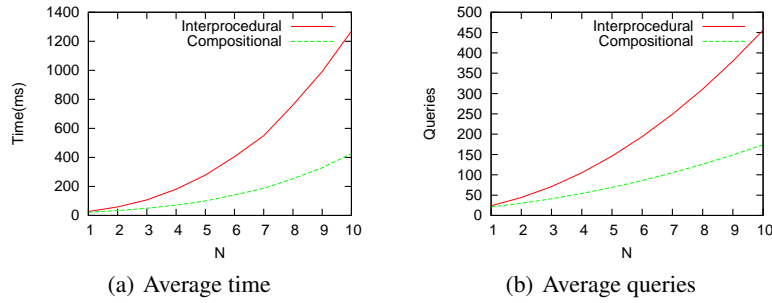


Figure 10(b) shows that the amount of queries performed by the interprocedural tool is quadratic in the bound N . The compositional tool clearly does less queries and is almost linear in N . It is not entirely linear due to re-evaluating unreachable equivalence classes. In Figure 10(a), one can see that the execution time is clearly less for the compositional tool. Although the compositional queries are more expensive than the interprocedural queries, the sheer number of queries causes the interprocedural tool to take more time.

7 Related work

Compositional symbolic execution was first introduced in the context of SMART [11], as an extension of the automatic testing tool DART [17]. The authors informally argue

that SMART is sound and complete (as a bugfinder) relatively to DART. In addition, DART is always sound (precise) and it is complete when it terminates [17]. The precision proofs depends critically on the dynamic aspect of SMART and DART. This paper only depends on the precision of the interpretation rules. When the interpretation rules are imprecise in SMART or DART, it either causes incompleteness or non-termination. In addition, the progress property is stronger than completeness upon termination.

With demand-driven compositional symbolic execution [12], the dependency on the depth-first search order of SMASH is lifted. To achieve this, function summaries are encoded in the SMT-solver. In addition, when no explored path to a point can be found, the algorithm constructs an indirect path that follows some unexplored path through the program. The authors claim relative completeness (as a bugfinder), and termination for programs with finite amounts of paths. The progress property in this paper is less algorithm specific and therefore more clear. In addition, it lifts the need for a termination argument. In the absence of fairness, demand-driven compositional symbolic execution does not satisfy the stronger progress property.

Finally, the system SMASH [13] combines the aspect of compositional analysis with may-must alternation. SMASH significantly outperforms both may-only, must-only and non-compositional may-must analysis. The analysis in this paper is a must analysis. As part of the soundness argument, the authors show that the must analysis of SMASH is precise. In addition, they show that the may analysis of SMASH is sound. Unfortunately, the combination of a sound may analysis with a precise must analysis is not necessarily a semi-decision procedure.

8 Conclusion

This paper creates a formal framework for compositional symbolic execution, based on a small but powerful calculus. We have modeled compositional symbolic execution as a transition system and formalized the meaning of precision and progress. In addition, we have proven that the algorithm is precise, and makes progress if the choices are fair. Finally, we have shown preliminary results of an implementation of the algorithm that is precise and progressing, and hence is a semi-decision procedure.

References

1. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7) (1976) 385–394
2. Tillmann, N., de Halleux, J.: Pexwhite box test generation for .net. In: *Proc. of Tests and Proofs '08*. Springer Berlin / Heidelberg (2008) 134–153
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: *Proc. of CCS '06*. (2006) 322–335
4. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: *NDSS, The Internet Society* (2008)
5. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The yogi project: Software property checking via static analysis and testing. In: *Proc. of TACAS '09*, Berlin, Heidelberg, Springer-Verlag (2009) 178–181

6. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of internet worms. *SIGOPS Oper. Syst. Rev.* **39**(5) (2005) 133–147
7. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D.: BitScope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University (March 2007)
8. Anand, S., Pasareanu, C.S., Visser, W.: Jpf-se: A symbolic execution extension to Java Pathfinder. In: *Proc. of TACAS 2007*, Braga, Portugal (March 2007) 134–138
9. Molnar, D.A., Wagner, D.: Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report 2007-23, University of California Berkeley (February 2007)
10. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: *Proc. of SIGSOFT '08/FSE-16*. (2008)
11. Godefroid, P.: Compositional dynamic test generation. In: *Proc. of POPL '07*. (2007) 47–54
12. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: *TACAS*. (2008) 367–381
13. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: unleashing the power of alternation. *SIGPLAN Not.* **45**(1) (2010) 43–56
14. European Computer Machinery Association: Standard ECMA-335: Common Language Infrastructure. 4th edition edn. (June 2006)
15. Evain, J.: Cecil. <http://www.mono-project.com/Cecil>
16. de Moura, L., Bjørner, N. In: *Z3: An Efficient SMT Solver*. Volume 4963/2008 of *Lecture Notes in Computer Science*. Springer Berlin (April 2008) 337–340
17. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. *SIGPLAN Not.* **40**(6) (2005) 213–223

Appendix

A Definition of rec and rer

In the remainder of this section, we defined the functions $rec(a, m)$ and $rer(a, m)$ which are used by the rules for call and return.

The function $rec(a, m)$ (Re-Evaluate Call) returns a new analysis state where all unreachable classes in each method which is reachable from m in the invocation graph are changed into unknown classes. A method n is reachable from a method m' in an set of invocations $invs$ (denoted $reachM(invs, m, m')$) if and only if $m = m'$ or there exists an invocation $\langle m, m'', \varsigma_G, pc \rangle \in invs$ such that m' is reachable from m'' in $invs$. The set of reachable methods from m in a set of invocations $invs$ denoted $rms(invs, m) = \{m' \in M_p \mid reachM(invs, m, m')\}$.

Formally:

$$rec(a, m) = \langle sum_a \oplus \bigcup_{m' \in rms(invs_a, m)} m' \mapsto rep(sum_a(m)), invs_a, rs_a \rangle$$

where $rep(\pi) = \{\epsilon \mid \epsilon \in \pi, state_\epsilon \neq unr\} \cup \{\langle unk, \nu_\epsilon, pc_\epsilon \rangle \mid \epsilon \in \pi, state_\epsilon = unr\}$ replaces all unreachable nodes by unknown nodes in the partition π .

The function $rer(a, m)$ (Re-Evaluate Return) returns a new analysis state where all unreachable classes that depend on the return of the method m are changed into unknown classes. A node depends on the return of m if its path condition depends on the return condition of m , or because it is reachable through an invocation where the path condition depends on the return condition of m .

A symbolic constraint pc depends on the return of a method m in the invocation graph $invs$ (denoted $dep(pc, m, invs)$) if and only if there exists an invocation $\langle m', m, \varsigma_G, pc \rangle \in invs$ such that pc contains the subterm $rc_m(\varsigma_G)$. A class ϵ or an invocation inv depends on the return of a method m in the invocation graph $invs$ (also denoted $dep(\epsilon, m, invs)$ and $dep(inv, m, invs)$) if and only if their symbolic constraint pc_ϵ or pc_{inv} depends on m in $invs$. The set $rmti(invs, m)$ of reachable methods through an invocation that depends on m in the invocation graph $invs$ is defined by $\{m' \mid m' \in M_p, inv \in invs, dep(inv, m, invs), reachM(invs, inv_{m_t}, m')\}$. Then $rer(a, m) = rer_2(rer_1(a, m), m)$ consists of two phases:

$$rer_1(a, m) = \langle sum_a \oplus \bigcup_{i \in calls(invs_a, m)} m_{s_i} \mapsto repd(sum_a(m_{s_i}), m_{s_i}), invs_a, rs_a \rangle$$

where $calls(invs, m)$ is the set of invocations to the method m in the current invocation graph $invs$, i.e. $calls(invs, m) = \{i \mid i \in invs, m_{s_i} = m\}$ and $repd(\pi, m, invs) = \{\epsilon \mid \epsilon \in \pi, state_\epsilon \neq unr \vee \neg dep(\epsilon, m, invs)\} \cup \{\langle unk, \nu_\epsilon, pc_\epsilon \rangle \mid \epsilon \in \pi, state_\epsilon = unr, dep(\epsilon, m, invs)\}$ re-evaluates the classes of π that depend on the return of m .

$$rer_2(a, m) = \langle sum_a \oplus \bigcup_{m' \in rmti(invs_a, m)} m' \mapsto rep(sum_a(m), inv), invs_a, rs_a \rangle$$